(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 930 793 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
21.07.1999 Bulletin 1999/29

(51) Int Cl.6: H04Q 7/32, H04B 1/38, G06F 9/38

(21) Application number: 98310312.8

(22) Date of filing: 16.12.1998

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: 22.12.1997 US 995606

(71) Applicant: TEXAS INSTRUMENTS INC.
Dallas, Texas 75243 (US)

(72) Inventors:
- McMahon, Michael (NMI)
  Plano, Texas 75074 (US)
- Lineberry, Marion C.
  Dallas, Texas 75218 (US)
- Woolsey, Matthews A.
  Plano, Texas 75023 (US)
- Chauvel, Gerard (NMI)
  06600 Antibes (FR)

(74) Representative: Potter, Julian Mark et al
D. Young & Co.,
21 New Fetter Lane
London EC4A 1DA (GB)

(54) **Mobile equipment with a plurality of processors**

(57) A wireless data platform (10) comprises a plurality of processors (12, 16). Channels of communication are set up between processors such that they may communicate information as tasks are performed. A dynamic cross compiler (80) executed on one processor compiles code into native processing code for another processor. A dynamic cross linker (82) links the compiled code for other processor. Native code may also be downloaded to the platform through use of a JAVA Bean (90) (or other language type) which encapsulates the native code. The JAVA Bean can be encrypted and digitally signed for security purposes.

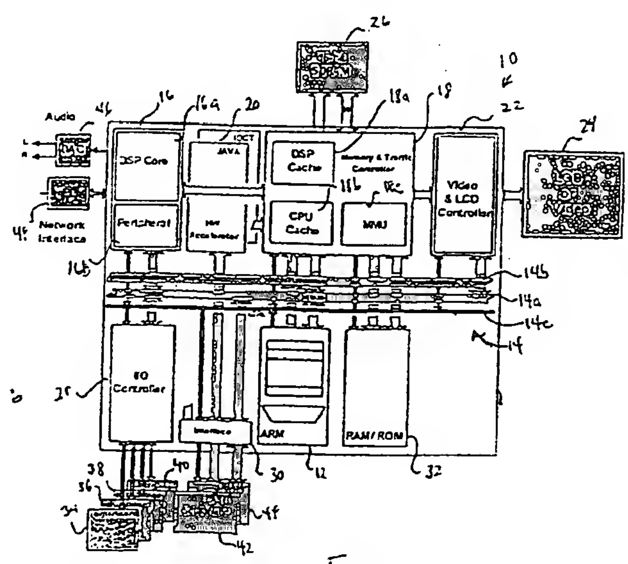

FIG 1

EP 0 930 793 A1

Description

BACKGROUND OF THE INVENTION

TECHNICAL FIELD

[0001] This invention relates in general to mobile electronic devices and, more particularly, to a hardware and software platform for mobile electronic devices.

DESCRIPTION OF THE RELATED ART

[0002] Handheld portable devices are gaining popularity as the power and, hence, functionality of the devices increases. Personal Digital Assistants (PDAs) are currently in widespread use and Smartphones, which combine some of the capabilities of a cellular phone and a PDA, are expected to have a significant impact on communications in the near future.

[0003] Some devices currently incorporate one or more DSPs (digital signal processor) or other coprocessors for providing certain discrete features, such as voice recognition, and a general purpose processor for other data processing functions. The code for the DSP and the code for the general purpose processor is generally stored in ROMs or other nonvolatile memories, which are not easily modified. Thus, as improvements and new features become available, it is often not possible to upgrade the capabilities of the device. In particular, it is not possible to maximize the use of the DSPs or other coprocessor which may be present in the device.

[0004] Therefore, a need exists for a data processing architecture which can be upgraded and optimizes use of multiple processors and coprocessors.

BRIEF SUMMARY OF THE INVENTION

[0005] The teachings of the present application disclose a mobile electronic device that comprises a coprocessor for executing native code, a host processor system operable to execute native code corresponding to the host processor system and processor independent code. The host processor system is operable to dynamically change the tasks performed by the digital signal coprocessor. Communication circuitry provides for communication between the host processor system and the coprocessor.

[0006] This mobile electronic device significant advantages over the prior art. Because the host processor system can dynamically allocate the tasks being performed by the coprocessor, which may be a digital signal processor, to fully use the coprocessor. The host processor system may direct a routine to one of a plurality of coprocessors, depending upon a variety of factors, such the present capabilities of each processor.

BRIEF DESCRIPTION OF SEVERAL VIEWS OF THE DRAWINGS

[0007] For a more complete understanding of the present invention, and the advantages thereof, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, in which:

Figure 1 illustrates a block diagram of a platform architecture particularly suited for general wireless data processing;

Figure 2 illustrates a functional block diagram of the platform of Figure 1;

Figure 3 illustrates a functional block diagram of dynamic cross compiling and dynamic cross linking functions;

Figure 4 illustrate an embodiment of native code for execution on a processor being encapsulated in a JAVA Bean wrapper for downloading to a device;

Figure 5 illustrates the operation of transferring the encapsulated native code to a processor on a device from a JAVA Bean located on a remote server; and

Figure 6 illustrates a flow diagram describing security features associated with the operation of Figure 5.

DETAILED DESCRIPTION OF THE INVENTION

[0008] Figure 1 illustrates a preferred embodiment of a general wireless data platform architecture, which could be used for example, in the implementation of a Smartphone or PDA. The wireless data platform 10 includes a general purpose (Host) processor 12 coupled to bus structure 14, including data bus 14a, address bus 14b and control bus 14c. One or more DSPs (or other coprocessors) 16, including the core processor 16a and the peripheral interface 16b, are coupled to bus 14 and to memory and traffic controller 18, which includes a DSP cache memory 18a, a CPU cache 18b, and a MMU (memory management unit) 18c. Hardware accelerator circuit 20 (for accelerating a portable language such as JAVA) and a video and LCD controller 22 are also coupled to the memory and traffic controller 18. The output of the video and LCD controller is coupled to an LCD or video display 24.

[0009] Memory & traffic controller 18 is coupled to bus 14 and to the main memory 26, shown as an SDRAM (synchronous dynamic random access memory). Bus 14 is also connected to I/O controller 28, interface 30, and RAM/ROM 32. A plurality of devices could be coupled to the wireless data platform 10, such as smartcard 34, keyboard 36, mouse 38, or one or more serial ports 40, such as a USB (universal serial bus) port or an RS232 serial port. Interface 30 can couple to a flash memory card 42 and/or a DRAM card 44. The peripheral interface 16b can couple the DSP 16 to a DAC (digital to analog converter) 46, a network interface 48 or to other devices.

[0010] The wireless data platform 10 of Figure 1 utilizes both a general purpose processor 12 and a DSP 16. Unlike current devices in which the DSP 16 is dedicated to specific fixed functions, the DSP 16 of Figure 1 can be used for any number of functions. This allows the user to derive the full benefit of the DSP 16.

[0011] One main area in which the DSP 16 can be used is in connection with the man-machine interface (MMI). Importantly, functions like speech recognition, image and video compression and decompression, data encryption, text-to-speech conversion, and so on, can be performed much more efficiently using the DSP 16. The proposed architecture allows new functions and enhancements to be easily added to wireless data platform 10.

[0012] It should be noted that the wireless data platform 10 is a general block diagram and many modifications could be made. For example, Figure 1 illustrates separate DSP and processor caches 18a and 18b. As would be known to one skilled in the art, a unified cache could also be used. Further, the hardware acceleration circuit 20 is an optional item. Such devices speed the execution of languages such as JAVA; however, the circuit is not necessary for operation of the device. Further, although the illustrated embodiment shows a single DSP, multiple DSPs (or other coprocessors) could be coupled to the buses.

[0013] Figure 2 illustrates a functional software architecture for the wireless data platform 10. This block diagram presumes the use of JAVA; it should be noted that languages other than JAVA could be used as well. Functionally, the software is divided into two groups, Host processor software and DSP software. The Host software includes one or more applets 41. The DSP API class 43 is a JAVA API package for JAVA applications or applets to access the functionality of the DSP API 50 and Host DSP Interface Layer 52. A JAVA virtual machine (VM) 45 interprets the applets. The JAVA native interface 47 is the method which the JAVA VM executes host processor or platform specific code. Native tasks 49 are non-JAVA programs which can be executed by the Host processor 12 without using the JAVA native interface. The DSP API 50, described in greater detail hereinbelow, is an API (application program interface) used the Host 12 to call to make use of the capabilities of the DSP 16. The Host-DSP Interface Layer 52 provides an API for the Host 12 and DSP 16 to communicate with each other, with other tasks, or other hardware using channels via the Host-DSP Communication Protocol. The DSP device driver 54 is the Host based device driver for the Host RTOS 56 (real time operating system) to communicate with the DSP 16. The Host RTOS 56 is an operating system, such as NUCLEUS PLUS by Accelerated Technology Incorporated.

[0014] Alternatively a non-real time operating system, such as WINDOWS CE by Microsoft Corporation, could be used. The DSP Library 58 contains programs stored for execution on the DSP 16.

[0015] On the DSP side, one or more tasks 60 can be stored in memory for execution by the DSP16. As described below, the tasks can be moved in and out of the memory as desired, such that the functionality of the DSP is dynamic, rather than static. The Host-DSP Interface layer 62 on the DSP side performs the same function as the Host-DSP Interface layer 52 on the Host side, namely it allows the Host 12 and DSP 16 to communicate. The DSP RTOS 64 is the operating system for the DSP processor. The Host Device driver 66 is a DSP based device driver for the DSP RTOS 64 to communicate with the Host 12. The Host-DSP Interface 70 couples the DSP 16 and Host 12.

[0016] In operation, the software architecture shown in Figure 2 uses the DSP 16 as a variable function device, rather than a fixed function device as in the prior art.

[0017] Accordingly, the DSP functions can be downloaded to the mobile device incorporating the architecture of Figure 2 to allow the DSP 16 to perform various signal processing functions for the Host 12.

[0018] The DSP-API provides a device independent interface from the Host 12 to the DSP 16. The functions provide the Host 12 with the ability to load and schedule tasks on the DSP 16 and to control and communicate with those tasks. The API functions include calls to: determine the DSP's available resources, create and control Host 12 and DSP tasks, create and control data channels between Host 12 and DSP tasks, and communicate with tasks. These functions are

described below. Each function returns a BOOLean result, which will be SUCCESS for a successful operation, or FAILURE. If the result is FAILURE, the errcode should be checked to determine which error occurred.

DSP_Get_MIPS

*BOOL DSP_Get_MIPS(T_DeviceID DevID, U32 *mips, U16 *errcode);*

[0019] This function returns the current MIPS available on the DSP. This consists of the MIPS capability of the DSP 16 minus a base MIPS value (the MIPS value with no additional dynamic tasks, i.e. the kernel plus API code plus drivers), minus the sum of the MIPS ratings for all loaded dynamic tasks. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING

DSP_Get_Memory_Available

*BOOL DSP_Get_Memory_Available(T_DeviceID DevID, T_Size *progmem, T_Size *datamem, U16 *errcode);*

[0020] This function will query the DSP 16 specified by *DevID* for the amounts of available memory for both program memory and data memory. The resultant values are returned in the *progmem* and *datamem* parameters. The sizes are specified in T_DSP_Words. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING

DSP_Alloc_Mem

*BOOL DSP_Alloc_Mem(T_DeviceID DevID, U16 mempage, T_Size size, T_DSP_Word **memptr, U16 *errcode);*

[0021] This function will allocate a block of memory on a DSP 16. The *DevID* specifies which device on which to allocate the memory. The *mempage* is 0 for program space, and 1 for data space. The *size* parameter specifies the memory block size in T_DSP_Words. The returned *memptr* will be a pointer to the memory block on the DSP 16, or NULL on failure. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_MEMPAGE
DSP_NOT_ENOUGH_MEMORY

DSP_Free_Mem

*BOOL DSP_Free_Mem(T_DeviceID DevID, U16 mempage, T_DSP_Word *memptr, U16 *errcode);*

[0022] This function will free a block of memory on a DSP that was allocated with the DSP_Alloc_Mem function. The *DevID* specifies on which device the memory resides. The *mempage* is 0 for program space, and 1 for data space. The *memptr* parameter is the pointer to the memory block. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_MEMPAGE
DSP_MEMBLOCK_NOT_FOUND

DSP_Get_Code_Info

*BOOL DSP_Get_Code_Info(char *Name, T_CodeHdr *codehdr, U16 *errcode);*

[0023] This function will access the DSP Library table and return the code header for the DSP function code specified by the *Name* parameter. On return, the location pointed to by the *codehdr* parameter will contain the code header information. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_NAMED_FUNC_NOT_FOUND

DSP_Link_Code

*BOOL DSP_Link_Code(T_DeviceID DevID, T_CodeHdr *codehdr, T_TaskCreate *tcs, U16 *errcode);*

[0024] This function will link DSP function code so that it will run at a specified address on the DSP specified by *DevID*. The *codehdr* parameter points to the code header for the function. The dynamic cross linker will link the code based on information in the code header, and in the code (COFF file). The dynamic cross linker will allocate the memory as needed, and link and load the code to the DSP 16. The *tcs* parameter is a pointer to the task creation structure needed in the DSP_Create_Task function. DSP_Link_Code will fill in the code entry points, priority, and quantum fields of the structure in preparation for creating a task. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS

DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_NOT_ENOUGH_PROG_MEMORY
DSP_NOT_ENOUGH_DATA_MEMORY
DSP_COULD_NOT_LOAD_CODE

DSP_Put_BLOB

*BOOL DSP_Put_BLOB(T_DeviceID DevID, T_HostPtr srcaddr, T_DSP_Ptr destaddr, U16 mempage, T_Size size, U16 *errcode);*

[0025] This function will copy a specified Binary Large Object (BLOB) to the DSP 16. The *DevID* specifies on which DSP 16 to copy the object. The *srcaddr* parameter is a pointer to the object in Host memory. The *destaddr* is a pointer to the location to which to copy the object on the DSP 16. The *mempage* is 0 for program space, and 1 for data space. The size parameter specifies the size of the object in T_DSP_Words. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_MEMPAGE

DSP_Create_Task

*BOOL DSP_Create_Task(T_DeviceID DevID, T_TaskCreate *tcs, T_TaskID *TaskID, U16 *errcode);*

[0026] DSP_Create_Task requests the DSP 16 to create a task given the task parameters and the code locations in the DSP's program space. The Task Creation Structure is show in Table 1:

Table 1.

Task Creation Structure.		
Data Type	**Field Name**	**Description**
T_DSP_Name	Name	User defined name for the task.
U32	MIPS	MIPS used by the task.
T_ChanID	ChanIn	The channel ID used for task input.
T_ChanID	ChanOut	The channel ID used for task output
T_StrmID	StrmIn	The stream ID used for task input
T_StrmID	StrmOut	The stream ID used for task output.
U16	Priority	The task's priority.
U32	Quantum	The task's timeslice in system ticks.
T_Size	StackReq	The amount of stack required.
T_DSP_Ptr	MsgHandler	Pointer to code to handle messages to the task.
T_HOST_Ptr	CallBack	Pointer to Host code to handle messages from the task.
T_DSP_Ptr	Create	Pointer to code to execute when task is created.
T_DSP_Ptr	Start	Pointer to code to execute when task is started.
T_DSP_Ptr	Suspend	Pointer to code to execute when task is suspended.
T_DSP_Ptr	Resume	Pointer to code to execute when task is resumed.
T_DSP_Ptr	Stop	Pointer to code to execute when task is stopped.

[0027] Once the task is created, the Create entry point will be called, giving the task the opportunity to do any necessary preliminary initialization. The Create, Suspend, Resume, and Stop entry points can be NULL. The resultant *TaskID* contains both the device ID (DevID), and the DSP's task ID. If the *TaskID* is NULL, the create failed. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_PRIORITY
DSP_CHANNEL_NOT_FOUND
DSP_ALLOCATION_ERROR

DSP_Start_Task

*BOOL DSP_Start_Task(T_TaskID TaskID, U16 *errcode);*

[0028] This function will start a DSP task specified by *TaskID*. Execution will begin at the task's Start entry point. The

errcode parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_TASK_NOT_FOUND

DSP_Suspend_Task

*BOOL DSP_Suspend_Task(T_TaskID TaskID, U16 *errcode);*

[0029] This function will suspend a DSP task specified by *TaskID*. Prior to being suspended, the task's Suspend entry point will be called to give the task a chance to perform any necessary housekeeping. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_TASK_NOT_FOUND

DSP_Resume_Task

*BOOL DSP_Resume_Task(T_TaskID TaskID, U16 *errcode);*

[0030] This function will resume a DSP task that was suspended by DSP_Suspend_Task. Prior to being resumed, the task's Resume entry point will be called to give the task a chance to perform any necessary housekeeping. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_TASK_NOT_FOUND
DSP_TASK_NOT_SUSPENDED

DSP_Delete_Task

*BOOL DSP_Delete_Task(T_TaskID TaskID, U16 *errcode);*

[0031] This function will delete a DSP task specified by *TaskID*. Prior to the deletion, the task's Stop entry point will be called to give the task a chance to perform any necessary cleanup. This should include freeing any memory that was allocated by the task, and returning any resources the task acquired. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_TASK_NOT_FOUND

DSP_Change_Task_Priority

*BOOL DSP_Change_Task_Priority(T_TaskID TaskID, U16 newpriority, U16 *oldpriority, U16 *errcode);*

[0032] This function will change the priority of a DSP task specified by *TaskID*. The priority will be changed to *newpriority*. The possible values of *newpriority* are RTOS dependent. Upon return, the *oldpriority* parameter will be set to the previous priority of the task. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_TASK_NOT_FOUND
DSP_INVALID_PRIORITY

DSP_Get_Task_Status

*BOOL DSP_Get_Task_Status(T_TaskID TaskID, U16 *status, U16 *priority, T_ChanID *Input, T_ChanID *Output, U16 *errcode);*

[0033] This function returns the status for a DSP task specified by *TaskID*. The *status* will be one of the following values:

DSP_TASK_RUNNING
DSP_TASK_SUSPENDED
DSP_TASK_WAITFOR_SEM
DSP_TASK_WAITFOR_QUEUE
DSP_TASK_WAITFOR_MSG

[0034] The *priority* parameter will contain the task's priority, and the *Input* and *Output* parameters will contain the task's input and output channel IDs, respectively. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND

DSP_DEVID_NOT_RESPONDING
DSP_TASK_NOT_FOUND

DSP_Get_ID_From_Name

*BOOL DSP_Get_ID_From_Name(T_DeviceID DevID, T_DSP_Name Name, T_DSP_ID *ID, U16 *errcode);*

[0035] This function returns the ID for a named object on the DSP 16. The named object may be a channel, a task, a memory block, or any other supported named DSP object. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_NAME_NOT_FOUND

DSP_Dbg_Read_Mem

*BOOL DSP_Dbg_Read_Mem(DEVICE_ID DevID, U8 mempage, DSP_PTR addr, U32 count, DSP_WORD *buf, U16 *errcode);*

[0036] This function requests a block of memory. The *mempage* specifies program memory (0) or data memory (1). The *addr* parameter specifies the memory starting address, and the *count* indicates how many T_DSP_Words to read. The bufparameter is a pointer to a caller provided buffer to which the memory should be copied. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_MEMPAGE

DSP_Dbg_Write_Mem

*BOOL DSP_Dbg_Write_Mem(T_DeviceID DevID, U16 mempage, T DSP_Ptr addr, T_Count count, T_DSP_Word *buf. U16 *errcode);*

[0037] This function writes a block of memory. The *mempage* specifies program memory (0) or data memory (1). The *addr* parameter specifies the memory starting address, and the *count* indicates how many T_DSP_Words to write. The *buf* parameter is a pointer the buffer containing the memory to write. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_MEMPAGE

DSP_Dbg_Read_Reg

*BOOL DSP_Dbg_Read_Reg(T_DeviceID DevID, U16 RegID, T_DSP_Word *regvalue, U16 *errcode);*

[0038] This function reads a DSP register and returns the value in *regvalue*. The *RegID* parameter specifies which register to return. If the *RegID* is -1, then all of the register values are returned. The *regvalue* parameter, which is a pointer to a caller provided buffer, should point to sufficient storage to hold all of the values. The register IDs are DSP specific and will depend on a particular implementation. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_REGISTER

DSP_Dbg_Write_Reg

*BOOL DSP_Dbg_Write_Reg(T_DeviceID DevID, U16 RegID, T_DSP_Word regvalue, U16 *errcode);*

[0039] This function writes a DSP register. The *RegID* parameter specifies which register to modify. *regvalue* contains the new value to write. The register IDs are DSP specific and will depend on a particular implementation. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_REGISTER

DSP_Dbg_Set_Break

*BOOL DSP_Dbg_Set_Break(T_DeviceID DevID, DSP_Ptr addr, U16 *errcode);* This function sets a break point at the given code address *(addr)*. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND

DSP_DEVID_NOT_RESPONDING

DSP_Dbg_Clr_Break

*BOOL DSP_Dbg_Clr_Break(T_DeviceID DevID, T_DSP_Ptr addr, U16 *errcode);*

[0040] This function clears a break point that was previously set by DSP_Dbg_Set_Break at the given code address *(addr)*. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_BP_DID_NOT_EXIST

[0041] The DSP Device Driver 54 handles communications from the Host 12 to the DSP 16. The driver functions will take the communication requests as specified in the Host-DSP Communications Protocol and handle the transmission of the information via the available hardware interface. The device driver is RTOS dependent and communications hardware dependent.

[0042] The DSP Library 58 contains the blocks of code that can be downloaded to the DSP 16 for execution. Each block of code will be previously unlinked, or relocatably linked as a library, so that the dynamic cross linker can resolve all address references. Each code block will also include information about the block's requirements for DSP MIPS (millions of instructions per second), priority, time slice quantum, and memory. The format for the code block header is shown in Table 2. The program memory and data memory sizes are approximations to give the Host 12 a quick check on whether the DSP can support the task's memory requirements. If there appears to be sufficient space, the dynamic cross linker can then attempt to link and load the code. It should be noted that the dynamic cross linker could still fail, due to page alignment and contiguity requirements. In the preferred embodiment, the code is in a version 2 COFF file format.

Table 2.

Code Block Header.		
Data Type	**Field Name**	**Description**
U16	Processor	The target processor type.
T_DSP_Name	Name	Task's name.
U32	MIPS	Worst case MIPS required by the task.
T_Size	ProgSize	Total program memory size needed.
T_Size	DataSize	Total data memory size needed.
T_Size	InFrameSize	Size of a frame in the task's input channel.
T_Size	OutFrameSize	Size of a frame in the task's output channel.
T_Size	InStrmSize	Size of the task's input stream FIFO.
T_Size	OutStrmSize	Size of the task's output stream FIFO.
U16	Priority	Task's priority.
U32	Quantum	Task's time slice quantum (number of system ticks).
T_Size	StackReq	Stack required.
T_Size	CoffSize	Total size of the COFF file.
T_DSP_Ptr	MsgHandler	Offset to a message handler entry point for the task.
T_DSP_Ptr	Create	Offset to a create entry point that is called when the task is created.
T_DSP_Ptr	Start	Offset to the start of the task's code.
T_DSP_Ptr	Suspend	Offset to a suspend entry point that is called prior to the task being suspended.
T_DSP_Ptr	Resume	Offset to a resume entry point that is called prior to the task being resumed.
T_DSP_Ptr	Stop	Offset to a stop entry point that is called prior to the task being deleted.
T_Host_Ptr	CoffPtr	Pointer to the location of the COFF data in the DSP Library.

[0043] A procedure for converting portable (processor independent) code, such as JAVA code, into linked target code is shown in Figure 3. The procedure uses two functions, a dynamic cross compiler 80 and a dynamic cross linker 82. Each function is implemented on the host processor 12. The dynamic cross linker is part of the DSP-API in the preferred embodiment. The cross compiler may also be part of the DSP-API.

[0044] The dynamic cross compiler 80 converts portable code into unlinked, executable target processor code. The dynamic cross linker 82 converts the unlinked, executable target processor code into linked, executable target processor code. To do so, it must resolve addresses within a block of code, prior to loading on the DSP 16. The dynamic

cross linker 82 links the code segments and data segments of the function, allocates the memory on the DSP 16, and loads the code and constant data to the DSP 16. The functions are referred to as "cross" compiling and "cross" linking, because the functions (compiling and linking) occur on a different processor (i.e., the host processor 12) from the target processor which executes the code (i.e., the DSP 16).

[0045] The dynamic cross compiler 80 accepts previously unlinked code loaded on demand by a user or a user agent (such as a browser). The code is processed to either (1) identify "tagged" sections of the code or (2) analyze untagged code segments for suitability of execution on the DSP 16. A tagged section of source code could delineate source targetable to a DSP by predetermined markers such as "<start DSP code>" and <end DSP code>" embedded in the source code. If a tagged section is identified either directly or through analysis, a decision is made to either cross compile or not based on the current processing state of the DSP 16. If a decision is made to compile, the section of code processed by compiling software that outputs unlinked, executable target processor code, using well known compiling methods. A decision not to compile could be made if for example, the DSP has insufficient available processing capacity (generally stated as available MIPS - million of instructions per second) or insufficient available memory, due to other tasks being executed by the DSP 16. The compiled code can be passed to the dynamic cross linker 82 for immediate use in the DSP 16, or could be saved in the DSP library 58.

[0046] The dynamic cross linker 82 accepts previously unlinked code, which is either (1) statically stored in connection with the host processor 12 or (2) dynamically downloaded to the host processor 12 over a network connection (including global networks such as the Internet) or (3) dynamically generated by the dynamic cross compiler 80. The dynamic cross linker 82 links the input code for a memory starting address of the DSP 16 determined at runtime. The memory starting address can be determined from a memory map or memory table stored on and managed by either the host processor 12 or DSP 16. The dynamic cross linker 82 convert referenced memory locations in the code to actual memory locations in the DSP 16. These memory locations could include, for example, branch addresses in the code or references to locations of data in the code.

[0047] In the preferred embodiment, the portable code is in a COFF (common object file format) which contains all information about the code, including whether it is linked or unlinked. If it is unlinked, symbol tables define the address which must be changed for linking the code.

[0048] The conversion process described above has several significant advantages over the prior art. First, the dynamic cross compiler 80 allows run-time decisions to be made about where to execute the downloaded portable code. For example, in a system with multiple target processors (such as two DSPs 16), the dynamic cross compiler 80 could compile the portable code to any one of the target processors based on available resources or capabilities. The dynamic cross linker 82 provides for linking code to run on a target processor which does not support relocatable code. Since the code is linked at run-time, memory locations in the DSP 16 (or other target processor) do not need to be reserved, allowing optimum efficiency of use of all computing resources in the device. Because the compiling is accomplished with knowledge of the architecture of the platform 10, the compiling can take advantage of processor and platform specific features, such as intelligent cache architectures in one or both processors 12 and 16.

[0049] Thus, the DSP 16 can have various functions which are changed dynamically to fully use its processing capabilities. For example, the user may wish to 12 load a user interface including voice recognition. At that time, the host processor 12 could download software and dynamically cross compile and cross link the voice recognition software for execution in the DSP 16. Alternatively, previously compiled software in the DSP library 58 could be dynamically cross linked, based on the current status of the DSP 16, for execution.

[0050] The Host Device Driver handles communications from the DSP 16 to the Host Processor 12. The driver functions takes the communication requests as specified in the Host-DSP Communications Protocol and handles transmission of the information via the available hardware interface. The device driver is RTOS dependent and communications hardware dependent.

[0051] The Host-DSP Communications Protocol governs the communications of commands and data between the Host 12 and the DSP 16. The communications consist of several paths: messages, data channels, and streams. Messages are used to send initialization parameters and commands to the tasks. Data channels carry large amounts of data between tasks and between the DSP 16 and Host 12, in the form of data frames. Streams are used to pass streamed data between tasks and between the DSP 16 and Host 12.

[0052] Each task has an entry point to a message handler, which handles messages. The messages are user defined and will include initialization parameters for the task's function, and commands for controlling the task. The tasks send messages to the Host 12 via the callback specified when the task is created. The prototype for the task's message handler and the prototype for the Host's callback are shown here:

```
void TaskMsgHandler(T_ReplyRef replyref, T_MsgID MsgID, T_Count count, T_DSP_Word *buf);
void HostCallBack(T_ReplyRef replyref T_MsgID MsgID, T_Count count, T_DSP_Word *buf);
```

[0053] The *replyref* parameter refers to an implementation dependent reference value, which is used to route the

reply back to the sender. For every Send_Message call, the recipient must call Reply_Message using the *replyref* parameter. The actual messages may appear as follows:

Sent message	MsgPktFlag	taskid	replyref	msgid	count	buf[.....]
Reply message	MsgPktFlag	-1	replyref	msgid	count	buf[.....]

The multiword data is sent least-significant word first.

[0054] A *TaskID* of 0 in the Send_Message function indicates a system level message. The system level messages are used to implement the DSP-API functions

[0055] Following are the Message functions:

Send_Message

*BOOL Send_Message(T_TaskID TaskID, T_MsgID MsgID, T_Count count, T_DSP_Word *msgbuf, T_DSP_Word *replybuf, T_Size replybufsize, T_Count replycount, U16 *errcode);*

[0056] This function will send a user defined message to a task specified by *TaskID*. The *MsgID* defines the message, and the *msgbuf* contains the actual message data. The message size is *count* T_DSP_Words. The reply *to* the message will be contained in the *replybuf* parameter, which points to a buffer of size *replybufsize*, provided by the caller. It should be of sufficient size to handle the reply for the particular message. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_TASK_NOT_FOUND

Reply_Message

*BOOL Reply_Message(T_ReplyRef replyref, T_Count count, T_DSP_Word *buf, U16 *errcode);*

[0057] This function is used to reply to messages. The *replyref* parameter is a reference used to route the reply back to the sender of the original message, and is implementation specific. The reply is contained in the *buf* parameter and its size is *count* T_DSP_Words. The *errcode* parameter will contain the following possible results:

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_BAD_REPLY_REF

[0058] The concept of channels is used to transmit frame-based data from one processor to another, or between tasks on the same processor. When created, a channel allocates a specified number and size of frames to contain the data. Initially, the channel will contain a list of empty frames. Tasks that produce data will request empty frames in which to put the data, then once filled, the frame is returned to the channel. Tasks that consume data will request full frames from the channel, and once emptied, the frame is returned to the channel. This requesting and returning of frame buffers allows data to move about with a minimum of copying.

[0059] Each task has a specified Input and Output channel. Once a channel is created, it should be designated as the input to one task, and the output to another task. A channel's ID includes a device ID, so channels can pass data between processors. Channel data flow across the Host-DSP interface may appear as follows:

ChanPktFlag	Channel ID	Count	Data[...]

The following are the channel functions:

Create_Channel

*BOOL Create_Channel(T_DeviceID DevID, T_Size framesize, T_Count numframes, T_ChanID *ChannelID, U16 *errcode);*

[0060] This function creates a data frame-based communication channel. This creates a channel control structure, which maintains control of a set of frame buffers, whose count and size are specified in the *numframes* and *framesize* parameters, respectively. When created, the channel allocates the data frames, and adds them to its list of empty frames. *ChannelID* will return the ID of the new channel. If the *DevID* is not that of the calling processor, a channel control structure is created on both the calling processor and the *DevID* processor, to control data flowing across the communications interface. The *errcode* parameter will contain the following possible results:

CHAN_SUCCESS
CHAN_DEVID_NOT_FOUND
CHAN_DEVID_NOT_RESPONDING

CHAN_ALLOCATION_ERROR

Delete_Channel

*BOOL Delete_Channel(T_ChanID ChannelID, U16 *errcode);*

[0061] This function deletes an existing channel specified by *ChannelID*. The *errcode* parameter will contain the following possible results:

CHAN_SUCCESS
CHAN_DEVID_NOT_FOUND
CHAN_DEVID_NOT RESPONDING
CHAN_CHANNEL_NOT_FOUND

Request_Empty_Frame

*BOOL Request_Empty_Frame(T_LocalChanID Chn, T_DSP_Word **bufptr, BOOL WaitFlag, U16 *errcode);*

[0062] This function requests an empty frame from the specified local channel ID. If *Chn* is NULL, then the task's output channel is used. Upon return, the *bufptr* parameter will contain the pointer to the frame buffer. If the *WaitFlag* is TRUE, and there is no frame buffer available, the caller will be suspended until a buffer becomes available. If the *WaitFlag* is FALSE, the function will return regardless. The *errcode* parameter will contain the following possible results:

CHAN_SUCCESS
CHAN_CHANNEL_NOT_FOUND
CHAN_BUFFER_UNAVAILABLE

Return_Full_Frame

*BOOL Return_Full_Frame(T_LocalChanID Chn, T_DSP_Word *bufptr, U16 *errcode);*

[0063] Once a task has filled a frame buffer, it returns is to the channel using this function. The buffer pointed to by *bufptr* is returned to the channel ID specified. If *Chn* is NULL, then the task's output channel is used. The *errcode* parameter will contain the following possible results:

CHAN_SUCCESS
CHAN_CHANNEL_NOT_FOUND
CHAN_BUFFER_CTRL_ERROR

Request_Full_Frame

*BOOL Request_Full_Frame(T_LocalChanID Chn, T_DSP_Word **bufptr. BOOL WaitFlag. U16 *errcode);*

[0064] This function requests a full frame of data from the specified local channel ID. If *Chn* is NULL, then the task's input channel is used. Upon return, the *bufptr* parameter will contain the pointer to the frame buffer. If the *WaitFlag* is TRUE, and there are no full frame buffers available, the caller will be suspended until a buffer becomes available. If the *WaitFlag* is FALSE, the function will return regardless. The *errcode* parameter will contain the following possible results:

CHAN_SUCCESS
CHAN_CHANNEL_NOT_FOUND
CHAN_BUFFER_UNAVAILABLE

Return_Empty_Frame

*BOOL Return_Empty_Frame(T_LocalChanID Chn, T_DSP_Word *bufptr, U16 *errcode);*

[0065] Once a task has used the data from a frame buffer, it should return the buffer to the channel using this function. The buffer pointed to by *bufptr* is returned to the channel ID specified. If *Chn* is NULL, then the task's input channel is used. The *errcode* parameter will contain the following possible results:

CHAN_SUCCESS
CHAN_CHANNEL_NOT_FOUND
CHAN_BUFFER_CTRL_ERROR

Set_Task_Input_Channel

*BOOL Set_Task_Input_Channel(T_Task *TaskID, T_ChanID ChanID, U16 *errcode);*

[0066] This function sets a task's input channel to the specified channel ID. The *errcode* parameter will contain the following possible results:

CHAN_SUCCESS
CHAN_DEVID_NOT_FOUND
CHAN_DEVID_NOT_RESPONDING
CHAN_TASK_NOT_FOUND
CHAN_CHANNEL_NOT_FOUND

Set_Task_Output_Channel

*BOOL Set_Task_Output_Channel(T_Task *TaskID, T_ChanID ChanID, U16 *errcode);*

[0067] This function sets a task's output channel to the specified channel ID. The *errcode* parameter will contain the following possible results:

CHAN_SUCCESS

CHAN_DEVID_NOT_FOUND
CHAN_DEVID_NOT_RESPONDING
CHAN_TASK_NOT_FOUND
CHAN_CHANNEL_NOT_FOUND

[0068] Streams are used for data, which can not be broken into frames, but which continuously flow into and out of a task. A stream will consist of a circular buffer (FIFO) with associated head and tail pointers to track the data as it flows in and out. Each task can have a designated input and output stream. Stream data flow across the Host-DSP interface may appear as follows:

StrmPktFlag	Stream ID	Count	Data[...]

Following are the stream functions:

Create_Stream

*BOOL Create_Stream(T_DeviceID DevID, T_Size FIFOsize, T_StrmID *StreamID, U16 *errcode);*

[0069] This function creates a FIFO-based communication stream. This creates a stream control structure, which maintains control of a FIFO of size *FIFOsize*. When created, the stream allocates an empty FIFO, and initializes head and tail pointers to handle data flow into and out of the stream. *StreamID* will return the ID of the new stream. If the *DevID* is not that of the calling processor, a stream control structure is created on both the calling processor and the *DevID* processor, to control data flowing across the communications interface. The *errcode* parameter will contain the following possible results:

STRM_SUCCESS
STRM_DEVID_NOT_FOUND
STRM_DEVID_NOT_RESPONDING
STRM_ALLOCATION_ERROR

Delete_Channel

*BOOL Delete_Stream(T_StrmID StreamID, U16 *errcode);*

[0070] This function deletes an existing stream specified by *StreamID*. The *errcode* parameter will contain the following possible results:

STRM_SUCCESS
STRM_DEVID_NOT_FOUND
STRM_DEVID_NOT_RESPONDING
STRM_STREAM_NOT_FOUND

Get_Stream_Count

*BOOL Get_Stream_Count(T_LocalStrmID StrmID, T_Count *count, U16 *errcode);*

[0071] This function requests the count of T_DSP_Words currently in the stream FIFO specified by *StrmID*. The *count* parameter will contain the number upon return. The *errcode* parameter will contain the following possible results:

STRM_SUCCESS
STRM_STREAM_NOT_FOUND

Write_Stream

*BOOL Write_Stream(T_LocalStrmID Strm, T_DSP_Word *bufptr, T_Count count, T_Count *countwritten, U16 *errcode);*

[0072] This function will write *count* number of T_DSP_Words to the stream specified by the *Strm*. If *Strm* is NULL, the task's output stream is used. The data is pointed to by the *bufptr* parameter. Upon return, *countwritten* will contain the number of T_DSP_Words actually written. The *errcode* parameter will contain the following possible results:

STRM_SUCCESS
STRM_DEVID_NOT_FOUND
STRM_DEVID_NOT_RESPONDING
STRM_STREAM_NOT_FOUND
STRM_STREAM_OVERFLOW

Read_Stream

*BOOL Read_Stream(T_LocalStrmID Strm, T_DSP_Word *bufptr, T_Count maxcount, BOOL WaitFlag, T_Count *countread, U16 *errcode);*

[0073] This function reads data from the stream specified by *Strm*. If *Strm* is NULL, the task's input stream is used. The data will be stored in the buffer pointed to by *bufptr*. Up to *maxcount* T_DSP_Words will be read from the stream. The *countread* parameter will contain the actual count of the data read. The *errcode* parameter will contain the following possible results:

STRM_SUCCESS

STRM_DEVID_NOT_FOUND
STRM_DEVID_NOT_RESPONDING
STRM_STREAM_NOT_FOUND

Set_Task_Input_Stream

*BOOL Set_Task_Input_Stream(T_Task *TaskID, T_StrmID StrmID, U16 *errcode);*

[0074] This function sets a task's input stream to the specified stream ID. The *errcode* parameter will contain the following possible results:

STRM_SUCCESS
STRM_DEVID_NOT_FOUND
STRM_DEVID_NOT_RESPONDING
STRM_TASK_NOT_FOUND
STRM_STREAM_NOT_FOUND

Set_Task_Output_Stream

*BOOL Set_Task_Output_Stream(T_Task *TaskID, T_StrmID StrmID, U16 *errcode);*

[0075] This function sets a task's output stream to the specified stream ID. The *errcode* parameter will contain the following possible results:

STRM_SUCCESS
STRM_DEVID_NOT_FOUND
STRM_DEVID_NOT_RESPONDING
STRM_TASK_NOT_FOUND
STRM_STREAM_NOT_FOUND

[0076] Data types used herein are defined in Table 3:

Table 3

Symbol	Description
S8	Signed 8-bit integer.
U8	Unsigned 8-bit integer.
S16	Signed 16-bit integer.
U16	Unsigned 16-bit integer.
S32	Signed 32-bit integer.
U32	Unsigned 32-bit integer.
T_HostWord	A word on the Host processor.
T_DSP_Word	A word on the DSP processor.
BOOL	Boolean value (TRUE or FALSE).
T_HostPtr	Pointer on the Host processor.
T_DSP_Ptr	Pointer on the DSP processor.
T_DeviceID	Processor device ID.
T_TaskID	A structure containing fields for a device ID and a processor local task ID.
T_ChanID	A structure containing fields for a device ID and a processor local channel ID.
T_MsgID	Message ID.
T_DSP_ID	An object ID on the DSP.
T_Count	Data type for a count.
T_Size	Data type for a size.
T_HostCallBack	Value used when tasks send message back to the Host.
T_ReplyRef	Message reply reference.
T_LocalTaskID	Local task ID.
T_LocalChanID	Local channel ID.

Table 3 (continued)

Symbol	Description
T_DSP_Name	Name for DSP objects (RTOS dependent).
T_CodeHdr	Code header structure for a DSP Library enlry.
T_TaskCreate	Task creation structure.

[0077] These tables define the messages passing between devices (i.e. Host to DSP 16). The device IDs present as parameters in the corresponding function calls are not incorporated in the messages since they are used to actually route the message to the device. Similarly, task IDs that include a device ID as their upper half for the function call will not include the device ID in the message, but only the DSP's local task ID portion.

Table 4

DSP-API Messages			
Message	Send Parameters	Reply Parameters	Direction Host ⇔ DSP
GET_MIPS	None	U32 mips	→
GET_MEM_AVAIL		T_Size progmem T_Size datamem	→
ALLOC_MEM	U16 mempage T_Size size	T_DSP_Word *memptr U16 errcode	→
FREE_MEM	U16 mempage T_DSP_Word *memptr	U16 errcode	→
PUT_BLOB	T_DSP_Ptr destaddr U16 mempage T_Size size T_DSP_Word BLOB[size]	U16 errcode	→
CREATE_TASK	T_TaskCreate tcs	T_TaskID TaskID U16 errcode	→
START_TASK	T_TaskID TaskID	U16 errcode	→
SUSPEND_TASK	T_TaskID TaskID	U16 errcode	→
RESUME_TASK	T_TaskID TaskID	U16 errcode	→
DELETE_TASK	T_TaskID TaskID	U16 errcode	→
CHANGE_PRIORITY	T_TaskID TaskID U16 newpriority	U16 oldpriority U16 errcode	→
GET_TASK_STATUS	T_TaskID TaskID	U16 status U16 priority T_ChanID Input T_ChanID Output U16 errcode	→
GET_ID	T_DSP_Name Name	T_DSP_ID ID U16 errcode	→

Table 5

DSP Interface Layer / Channel Interface Layer Messages			
Message	Send Parameters	Reply Parameters	Direction Host ⇔ DSP
CREATE_CHANNEL	T_Size framesize T_Count numframes	T_ChanID ChannelID U16 errcode	→

Table 5 (continued)

DSP Interface Layer / Channel Interface Layer Messages			
Message	Send Parameters	Reply Parameters	Direction Host ⇔ DSP
DELETE_CHANNEL	T_ChanID ChannelID	U16 errcode	→
CREATE_STREAM	T_Size FIFOsize	T_StrmID StreamID U16 errcode	→
DELETE_STREAM	I_StrmID StreamID	U16 errcode	→

Table 6

Debug Messages			
Message	Send Parameters	Reply Parameters	Direction Host ⇔ DSP
READ_MEM	U16 mempage T_DSP_Ptr addr T_Count count	T_DSP_Word mem[count] U16 errcode	→
WRITE_MEM	U16 mempage T_DSP_Ptr addr	U16 errcode	→
	T_Count count T_DSP_Word mem[count]		
READ_REG	U16 RegID	DSP_WORD regvalue U16 errcode	→
WRITE_REG	U16 RegID T_DSP_Word regvalue	U16 errcode	→
SET_BREAK	T_DSP_Ptr addr	U16 errcode	→
CLR_BREAK	T_DSP_Ptr addr	U16 errcode	→
BREAK_HIT	T_DSP_Ptr addr	U16 ACK	←

[0078] Figures 4 - 6 illustrate an embodiment for downloading native code to a target processor (i.e., the host 12 or DSP 16) in a secure and efficient manner. This embodiment for downloading code could be used, for example, in downloading code from the Internet, or other global network, from a Local or Wide Area Network, or from a peripheral device, such as a PC Card or Smartcard.

[0079] In Figure 4, an embodiment of a JAVA Bean 90 is shown, where the Bean 90 acts as a wrapper for native code 92. The Bean further includes several attributes 94, listed as a Code Type attribute 94a, a Code Size attribute 94b and a MIPS Required attribute 94c. The Bean 90 has several actions 96, including a Load Code action 96a, a Load Parameters action 96b and an Execute Parameter 96c.

[0080] In operation, the Load Code action 96a is used to load external native code (native to the target processor) into the Bean. Since JAVA Beans have persistence, the Bean 90 can store its internal state, including the native code 92 and the attributes 94. The Load Parameters action 96b retrieves parameters from the native code 92 (using, for example, the COFF file format described above) and stores the parameters as attributes 94a-c. The Execute action 96c executes tasks installed in the DSP 16.

[0081] Figure 5 illustrates use of the Bean 90 to download code to the target processor. In this example, it is assumed that the target processor is the DSP 16 (or one of multiple DSPs 16), although it could be used to download native code to the host processor 12 as well. Further, it is assumed that the desired Bean 90 is resident in a network server, such as a LAN server or an Internet server, although the Bean could be resident in any device in communication with the platform 10, such as a Smartcard. For a wireless data platform 10, the connection to the network server 100 will often be wireless.

[0082] In Figure 5, the platform 10 is coupled to a network server 100. The host processor 12, as shown in greater detail in Figure 2, may execute one or more JAVA applets 41 through a JAVA virtual machine 45. In order to download new code, the host 12 loads an applet 41 containing the Bean 90 from the network server 100 or the Bean, without the containing applet, can be downloaded from the server 100. Once the wrapper Bean 90 has been retrieved, it can be queried for the size of the native code, code type (for which processor is the code intended) and MIPs required. If

the intended processor has sufficient resources to run the code 92, the code 92 can be installed to execute on the intended processor, either the host processor 12 or DSP 16 in the architecture shown in Figure 5. Typically, the native code 92 will be unlinked, compiled code. Thus, the cross linker 82 of the DSP-API 50 will link the code to an available memory location. The Bean would pass the binary native code 92 to the dynamic cross linker 82, which would install and execute the code.

[0083] A typical manner in which a download of native code might occur is when the user is running an applet 41 in which a DSP function is desired. First, the applet 41 would check to see if the desired code was installed as a task 60 in the DSP or was available in the DSP Library 58. If so, the task could be executed without a download.

[0084] If the task is not stored in the DSP 16 or the DSP library 58, an object (referred to as the "DSPLoader" object herein) could be created to load the Bean 90. If the DSPLoader class is local on the host 12, JAVA will check to see if the Bean is available locally as well. In a first instance, there may be a Bean with the code stored locally. If so, the code from the Bean is installed to the DSP 16 (or to whichever processor specified by the Code Type). If a Bean without the code is stored locally, the Bean can retrieve the code from the appropriate server.

[0085] On the other hand, if the DSPLoader object is not local, then JAVA will load the Bean 90 from the server that wrote the applet 41. The code from the Bean will then be installed as described above.

[0086] While the downloading of native code is described in connection with the use of a JAVA Bean, it could also be accomplished by wrapping the code within another language, such as an ActiveX applet.

[0087] Using a JAVA Bean (or other applet) as a wrapper to the native code has significant advantages. First, it allows a simple, standard method for loading code onto one of a plurality of processors. The Bean is created, code is loaded into the Bean and the code is linked to the appropriate processor. Without wrapping the code within the Bean, the process may take several hundred steps. Second, it allows multiple pieces of native code to be combined by a single applet, providing for complex applications to be generated from multiple discrete routines using a single applet to combine the routines as desired. Third, it takes advantage of the language's security features, thereby protecting not only the JAVA code in the Bean 90, but the native code 92 as well. Other languages, such as ActiveX, have security features as well.

[0088] Two of the most important security features are digital signing and encryption. A JAVA Bean or ActiveX applet may be signed by the source of the code; when the Bean or applet is downloaded, the signature is verified by the receiving application, which has a list of trusted sources. If the Bean or applet is signed by a trusted source, it can be decrypted using standard techniques. Accordingly, the native code is encrypted during transmission along with the code of the Bean or applet, preventing unauthorized modification of the code. Because the native code is secure and comes from a trusted source, the attributes can also be trusted as accurate.

[0089] Figure 6 illustrates a flow chart describing the process of downloading native code for a processor using a JAVA Bean, it being understood that the native code could be wrapped in an applet of a different language using similar techniques. In step 110, the encrypted, digitally signed Bean 90 is downloaded to a device running a JAVA virtual machine. In step 112, the signature is verified. If it is not from a source listed as a trusted source, exception processing is enabled in step 114. In the case of the Bean coming from a trusted source, the exception processing function may give the user an opportunity to accept the Bean, if the user is comfortable with the source. If the signature is invalid, the exception processing may delete the Bean 90 and send an appropriate error message to the user.

[0090] If the signature is valid and comes from a trusted source, the Bean is decrypted in step 116. This step decrypts both the JAVA code and the native code in the Bean. In step 118, the attributes are retrieved from the Bean 90 and in step 120 the applet determines whether the appropriate processor has sufficient resources to run the code. If not, the exception processing step 114 may decline to install the native code, or steps may be taken to free resources. If there are sufficient resources, the code is linked using the cross-linker and installed on the desired processor in step 122. In step 124, the native code is executed.

[0091] Sample JAVA script for a Bean 90 is provided hereinbelow:

```
package ti.dsp.loader;

import java.awt.*;
import java.io.*;
import java.net.*;

public class NativeBean extends Canvas implements Serializable
{
    public NativeBean() {

        setBackground(Color.white);

        funcData = new ByteArrayOutputStream();

        try {
            funcCodeBase = new URL("http://localhost");
        }
        catch (MalformedURLException e) {
```

```
        }
    }

    public Dimension getMinimumSize()  {

        return new Dimension(50, 50);
    }

    public void loadCode()  {

        URL baseURL = null;

        try  {
            baseURL = new URL(funcCodeBase.toString() + "/" + myFunction);
        }
        catch (MalformedURLException e)  {
        }

        DataInputStream source = null;
        int read;
        byte[] buffer;

        buffer = new byte[1024];
        try  {
            source = new DataInputStream(baseURL.openStream());
        }
        catch (IOException e)  {
            System.out.println("IOException creating streams: " + e);
        }

        codeSize = 0;

        funcData.reset();

        try  {
            while (true)  {

                read = source.read(buffer);

                if (read == -1)
                    break;

                funcData.write(buffer, 0, read);
            }
        }
        catch (IOException e)  {
            System.out.println("IOException: " + e);
        }
```

```
        codeSize = funcData.size();
        System.out.println("Code size = " + codeSize);

        try {
            source.close();
        }
        catch (IOException e) {
            System.out.println("IOException closing: " + e);
        }
    }

    public synchronized String getFunctionName() {

        return myFunction;
    }

    public void setFunctionName(String function) {

        myFunction = function;
    }

    public synchronized String getCodeBase() {

        return funcCodeBase.toString();
    }

    public void setCodeBase(String newBase) {

        try {
            funcCodeBase = new URL(newBase);
        }
        catch (MalformedURLException e) {
        }
    }

    public void installCode() {

        FileOutputStream destination = null;
        File libFile = new File(myFunction);

        try {
            destination = new FileOutputStream(libFile);
        }
        catch (IOException e) {
            System.out.println("IOException creating streams: " + e);
        }

        if (destination != null) {
```

```
                try {
                    funcData.writeTo(destination);
                }
                catch (IOException e) {
                    System.out.println("IO Exception installing native code: " + e);
                }
            }
        }
            linkCode(funcData)

        public void loadParameters() {
        }

        public void execute() {
        }

        public synchronized int getCodeSize() {

            return codeSize;
        }

        public synchronized int getCodeType() {

            return codeType;
        }

        public void setCodeType(int newType) {

            codeType = newType;
        }

        private int codeSize = 0;
        private int codeType = 1;
        private String myFunction = "";
        private URL funcCodeBase = null;
        private ByteArrayOutputStream funcData = null;
    }
```

[0092] In the script set forth above, the NativeBean() routine creates the Bean 90 which will hold the native code. The loadCode() routine gets the native code from the server. The getFunctionName() and getCodeBase() routines retrieve attributes. The installCode() routine calls the cross linker to link the native code to the DSP and to load the linked code. The loadParameters() routine instructs the Bean to examine the native code and determine its attributes. The getCodesize() and getCodetype() routines transfer the attributes to the requesting applet.

[0093] Although the teachings disclosed herein have been directed to certain exemplary embodiments, various modifications of these embodiments, as well as alternative embodiments, will be suggested to those skilled in the art.

[0094] Further and particular embodiments of the invention will now be enumerated with reference to the following numbered clauses.

1. A mobile electronic device, comprising:

a coprocessor for executing native code;
a host processor system operable to execute native code corresponding to the host processor system and processor independent code, said host processor system operable to dynamically change the tasks performed

by the digital signal coprocessor; and
circuitry for communicating between said host processor system and said coprocessor.

2. The mobile electronic device of clause 1 and further comprising network interface circuitry for receiving data from a network.

3. The mobile electronic device of clause 2 wherein said network interface circuitry comprises wireless network circuitry.

4. The mobile electronic device of clause 3 wherein said network interface circuitry comprises circuitry for interfacing with a global network.

5. A method of controlling a mobile electronic device comprising the steps of:

executing native code in a coprocessor;
executing both native code and processor independent code in a host processor system;
dynamically changing the tasks performed by the digital signal coprocessor with said host processor system; and
communicating between said host processor system and said coprocessor.

6. The method of clause 5 and further comprising the step of receiving code through a network interface.

7. The method of clause 6 and further comprising the step of receiving code through a wireless network interface.

8. The method of clause 6 or 7 and further comprising the step of receiving code through a wireless network interface from a global network.

9. A mobile electronic device, comprising:

a plurality of coprocessors;
a host processor system operable to:

execute source code;
identify one or more sections of source code to be executed on one or more of said coprocessors; and
for each identified section of source code. determining a corresponding coprocessor: and
for each identified section of source code, compile said identified section of code into the native code associated with said corresponding coprocessor and install said native code onto said corresponding coprocessor; and

circuitry for communicating between said host processor system and said coprocessors.

10. The mobile electronic device of clause 9 wherein one or more of said coprocessors comprise digital signal processors.

Claims

1. A mobile electronic device, comprising:

a coprocessor for executing native code;
a host processor operable to execute native code corresponding to the host processor and processor independent code, said host processor operable to dynamically change the tasks performed by the digital signal coprocessor; and
circuitry for communicating between said host processor and said coprocessor.

2. The mobile electronic device of Claim 1, wherein said coprocessor comprises a digital signal processor.

3. The mobile electronic device of Claim 1 or Claim 2, wherein said processor independent code comprises JAVA.

4. The mobile electronic device of any preceding claim , wherein said host processor system is arranged to generate native code for said coprocessor.

5. The mobile electronic device of any preceding claim, wherein said host processor is arranged to generate native code for said coprocessor by compiling processor independent source code.

6. The mobile electronic device of any preceding claim, wherein said host processor is arranged to compile identified blocks of source code.

7. The mobile electronic device of any preceding claim, wherein said host processor system is arranged to identify blocks of source code that could be executed on the coprocessor and to compile said blocks of code.

8. The mobile electronic device of any preceding claims, further comprising:
   a memory for storing a library of routines that can be downloaded to said coprocessor for execution.

9. The mobile electronic device of any preceding claim further comprising a hardware language accelerator.

10. The mobile electronic device of any preceding claim wherein said hardware accelerator comprises a JAVA accelerator.

11. The mobile electronic device of any preceding claim further comprising network interface circuitry for receiving data from a network.

12. A method of controlling a mobile electronic device comprising of:

    executing native code in a coprocessor:
    executing both native code and processor independent code in a host processor
    dynamically changing the tasks performed by the digital signal coprocessor with said host processor and communicating between said host processor system and said coprocessor.

13. The method of claim 12 wherein said step of executing native code in a coprocessor comprises executing native code in a digital signal processor.

14. The method of claims 12 and 13 further comprising generating native code for coprocessor in said general processing system.

15. The method of claim 14 wherein said step of generating native code comprises the step of generating native code by compiling processor independent source code.

16. The method of any of claims 12 to 15 further comprising identifying blocks of said source code to compile for execution on said coprocessor.

17. The method of any of claims 12-16 further comprising storing a library of routines for downloading from said host processor system to said coprocessor for execution.

18. A mobile electronic device, comprising:

    a plurality of coprocessors;
    a host processor system operable to:

        execute source code;
        identify one or more portions of source code to be executed on one or more of said coprocessors; and
        for each identified portion of source code, determining a corresponding coprocessor; and
        for each identified portion of source code, compile said identified portion of code into the native code associated with said corresponding coprocessor and install said native code onto said corresponding coprocessor; and

    circuitry for communicating between said host processor system and said coprocessors.

19. A method ol controlling a mobile electronic device, comprising:

executing source code on a host processor system;
identifying one or more portions of source code to be executed on one or more coprocessors; and
for each identified portion of source code, determining a corresponding coprocessor; and
for each identified portion of source code, compiling said identified portion of code into the native code associated with said corresponding coprocessor and installing said native code onto said corresponding coprocessor; and
communicating between said host processor system and said coprocessors.

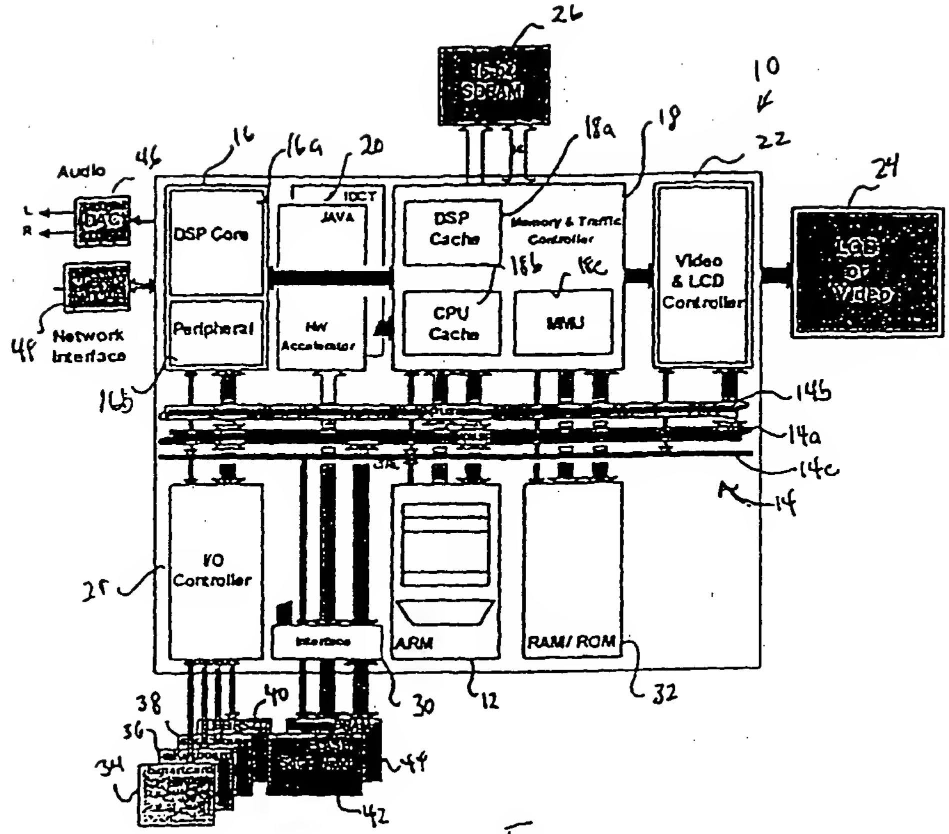
26
SDRAM
10
18a
18
22
24
Audio
46
16
16a
20
IDCT
JAVA
DSP Cache
Memory & Traffic Controller
L
R
DAC
DSP Core
18b
18c
Video & LCD Controller
LCD or Video
Peripheral
HW Accelerator
CPU Cache
MMU
Network Interface
48
16b
14b
14a
14c
14
I/O Controller
28
Interface
ARM
RAM/ROM
30
12
32
38
40
36
34
44
42

FIG 1

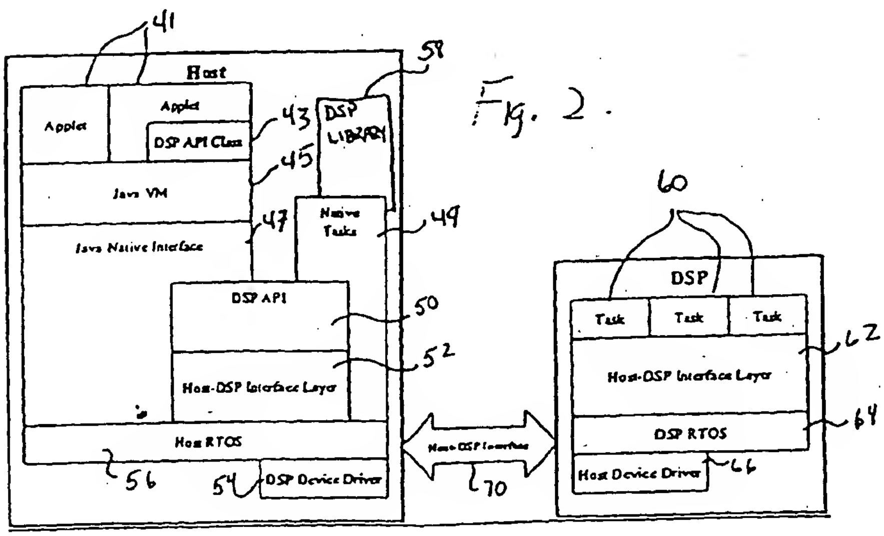
41
Host
58
Applet
Applet
DSP API Class
43
DSP LIBRARY
45
Java VM
Native Tasks
47
49
Java Native Interface
DSP API
50
Host-DSP Interface Layer
52
Host RTOS
56
54
DSP Device Driver
Host-DSP Interface
70
60
DSP
Task
Task
Task
Host-DSP Interface Layer
62
DSP RTOS
64
Host Device Driver
66

Fig. 2.

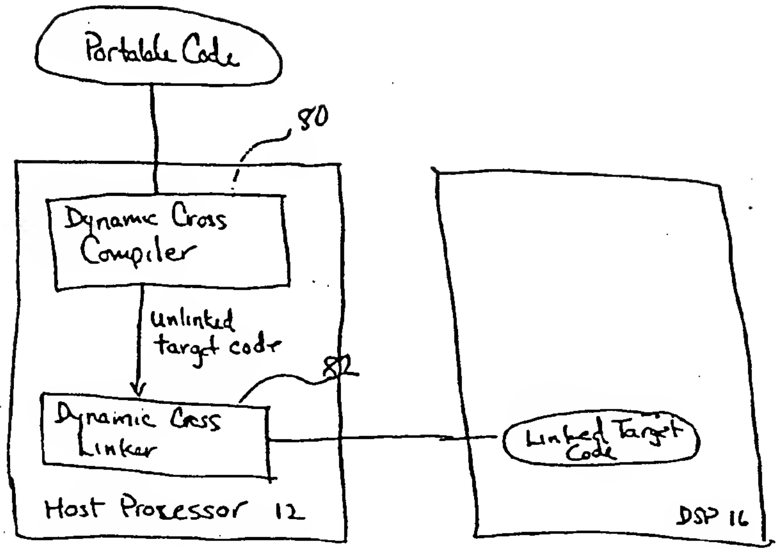

Figure 3

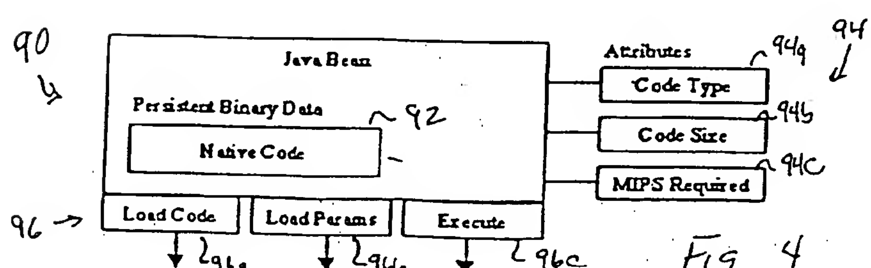

Fig 4

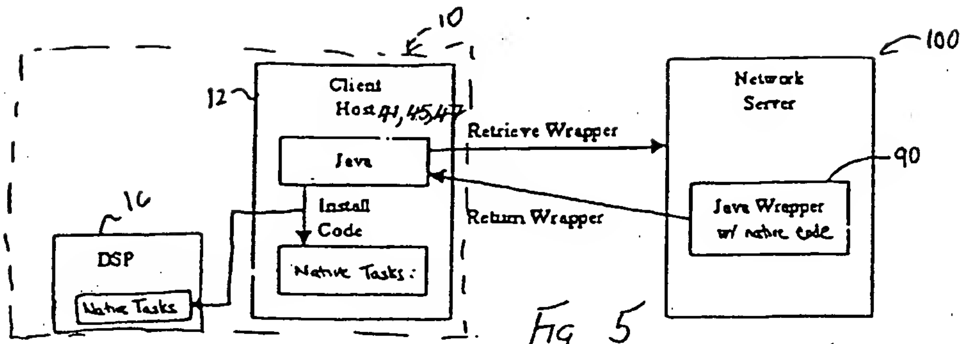
10
100
12
Client
Host 41, 45, 47
Retrieve Wrapper
Network
Server
Java
90
16
Install
Code
Return Wrapper
Java Wrapper
w/ native code
DSP
Native Tasks
Native Tasks

Fig 5

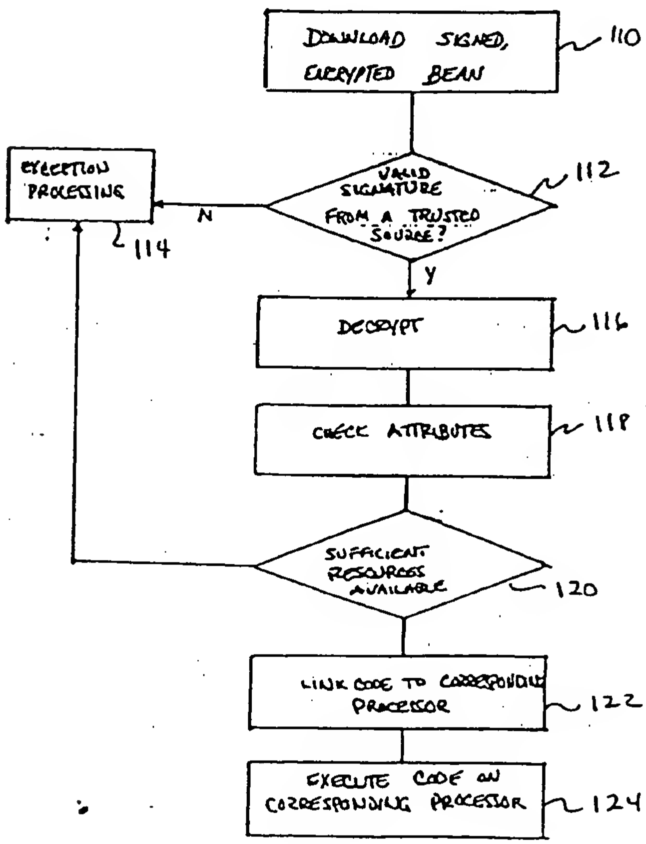
DOWNLOAD SIGNED,
ENCRYPTED BEAN
110
EXCEPTION
PROCESSING
VALID
SIGNATURE
FROM A TRUSTED
SOURCE?
112
N
114
Y
DECRYPT
116
CHECK ATTRIBUTES
118
SUFFICIENT
RESOURCES
AVAILABLE
120
LINK CODE TO CORRESPONDING
PROCESSOR
122
EXECUTE CODE ON
CORRESPONDING PROCESSOR
124

FIG. 6

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 98 31 0312

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
P,X	WO 98 40978 A (SAGEM ;DEMEURE JEAN ANDRE (FR); DIMECH JEAN MARC (FR)) 17 September 1998 * page 4, line 22 - line 27 * * page 5, line 25 - line 28 * * page 8, line 26 - line 29 *	1,2,4, 11,12, 18,19	H04Q7/32 H04B1/38 G06F9/38
P,X	EP 0 869 691 A (DEUTSCHE TELEKOM AG) 7 October 1998 * column 2, line 4 - line 22 *	1-4, 11-14, 18,19	
A	GB 2 310 575 A (WESTINGHOUSE ELECTRIC CORP) 27 August 1997 * page 5, line 16 - line 25 *	1,2,12, 18,19	
A	WO 97 26750 A (CELLPORT LABS INC) 24 July 1997 * page 18, line 6 - page 22, line 26 *	1,12,18, 19	
A	US 4 862 407 A (FETTE BRUCE A ET AL) 29 August 1989 * column 4, line 49 - line 58 * * column 13, line 14 - line 18 *	1,12,18, 19	TECHNICAL FIELDS SEARCHED (Int.Cl.6) H04Q H04M G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
BERLIN	31 May 1999	Leouffre, M

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 98 31 0312

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

31-05-1999

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 9840978 A	17-09-1998	FR 2760917 A	18-09-1998
		FR 2760918 A	18-09-1998
		AU 6921998 A	29-09-1998
EP 0869691 A	07-10-1998	DE 19713965 A	08-10-1998
GB 2310575 A	27-08-1997	AU 1264397 A	28-08-1997
WO 9726750 A	24-07-1997	US 5732074 A	24-03-1998
		AU 1525197 A	11-08-1997
		CA 2243454 A	24-07-1997
		EP 0875111 A	04-11-1998
US 4862407 A	29-08-1989	NONE	

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82